qvm-remote: Authenticated Remote Execution in Qubes OS dom0 via File-Based Queues

A Pull-Model RPC Framework with HMAC-SHA256
Authentication and Full Audit Trails

Gabriele Risso
`gabri.risso@gmail.com`
https://github.com/GabrieleRisso/qvm-remote

February 2026

Abstract

Qubes OS enforces strict isolation by prohibiting code execution from virtual machines (VMs) in the privileged `dom0` domain. While essential for security, this restriction creates significant friction for developers, system administrators, and automation pipelines that need to manage VMs programmatically. We present **qvm-remote**, an open-source tool that provides "SSH for dom0"—authenticated remote command execution from VMs to `dom0` through a file-based queue protocol. The system uses a pull model where `dom0` initiates all I/O (never the VM), HMAC-SHA256 authentication with 256-bit per-VM keys, comprehensive input validation, and dual-sided audit logging. We describe the protocol design, analyze its security properties against the Qubes threat model, present a formal analysis of the authentication scheme, and demonstrate that the framework adds <50 ms overhead per command while providing cryptographic guarantees against replay, forgery, and cross-VM attacks. qvm-remote is implemented in pure Python 3 (stdlib only) and is packaged for Fedora RPM, Arch Linux, and the Qubes Builder v2 build system.

Keywords: Qubes OS, dom0, remote execution, HMAC-SHA256, file-based queue, pull model, qrexec, system administration, Xen

1 Introduction

Qubes OS [1] implements a security-by-compartmentalization architecture where the privileged `dom0` domain manages all VMs but is intentionally isolated from them. This design prevents a compromised VM from executing code in `dom0`—a critical security property, since `dom0` has unrestricted access to all VM memory, storage, and configuration.

However, this same property creates a significant usability barrier for legitimate administration tasks. Consider a developer working in a `code` VM who needs to:

- Resize another VM's memory: `qvm-prefs work memory 4096`
- List running VMs: `qvm-ls -running`
- Deploy a service: `systemctl start my-tunnel`
- Query hypervisor state: `xl info`

Each of these requires physically switching to the `dom0` terminal, typing the command, and switching back—a context switch that breaks flow and discourages automation.

qvm-remote bridges this gap with a carefully designed protocol that provides VM-to-dom0 command execution while preserving auditability and cryptographic authentication. The key design principle is the *pull model*: `dom0` always initiates I/O operations, never the VM. The VM merely writes requests to its own local filesystem; `dom0` discovers and processes them at its own pace.

1.1 Contributions

- A **file-based queue protocol** for VM-to-dom0 RPC that preserves the pull-model invariant

(Section 3).

- An **HMAC-SHA256 authentication scheme** with per-VM keys and per-command tokens that prevents forgery, replay, and cross-VM attacks (Section 4).
- A **defense-in-depth security model** with input validation, execution sandboxing, and dual-sided audit trails (Section 5).
- A **production implementation** in pure Python 3 with zero external dependencies, packaged for RPM, Arch, Salt, and Qubes Builder (Section 6).

2 Background and Motivation

2.1 The dom0 Isolation Principle

In the Qubes security model, `dom0`:

1. Has **no network interface**—immune to remote attack.
2. Runs the **Xen toolstack** (`xl`, `libvirt`).
3. Hosts the **GUI compositor** (rendering VM windows).
4. Manages **qrexec policies**—the inter-VM firewall.

Any code executing in `dom0` runs with the privilege level of the hypervisor management plane. This is why Qubes treats `dom0` as sacrosanct: no VM should be able to cause code execution there.

2.2 Existing Approaches

Manual switching: The default. Secure but disruptive to workflow. Incompatible with automation.

Custom qrexec services: Each command requires a separate `/etc/qubes-rpc/` handler and policy. Scales poorly for ad-hoc commands.

qubes-remote (v0.x): An earlier bash-based prototype by the same author. Limited error handling, no HMAC auth, no audit trail, no input validation.

qvm-remote addresses all of these limitations with a unified, authenticated, and auditable framework.

3 Protocol Design

3.1 The Pull-Model Invariant

The fundamental design constraint is:

> *dom0 initiates every I/O operation. The VM never pushes data to dom0; it only writes to its own local filesystem.*

This preserves the Qubes principle that VMs cannot cause effects in `dom0`—`dom0` *chooses* to read from the VM. The distinction is subtle but important: the VM's "request" is a passive file on its own disk, not an active network connection or syscall into `dom0`.

3.2 Queue Structure

Each authorized VM maintains a queue directory:

```
~/.qvm-remote/
  queue/
    pending/ # new commands
      20260218-143022-1234-a1b2c3d4
      20260218-143022-1234-a1b2c3d4.auth
    running/ # in-progress (moved by dom0)
    results/ # completed
      <cmd_id>.out # stdout
      <cmd_id>.err # stderr
      <cmd_id>.exit # exit code
      <cmd_id>.meta # timing metadata
  auth.key # 256-bit HMAC key (0600)
  audit.log # VM-side audit trail
  history/ # archived results
    2026-02-18/
      <cmd_id>/
```

3.3 Command Lifecycle

The protocol proceeds in five phases:

1. **Enqueue** (VM): The client generates a unique command ID ($\mathit{cid} = \text{timestamp-pid-random}_8$), writes the command body to `pending/`*`cid`*, and writes HMAC-SHA256(k, cid) to `pending/`*`cid`*`.auth`.
2. **Poll** (dom0): The daemon lists `pending/` via `qvm-run -pass-io -no-autostart`.

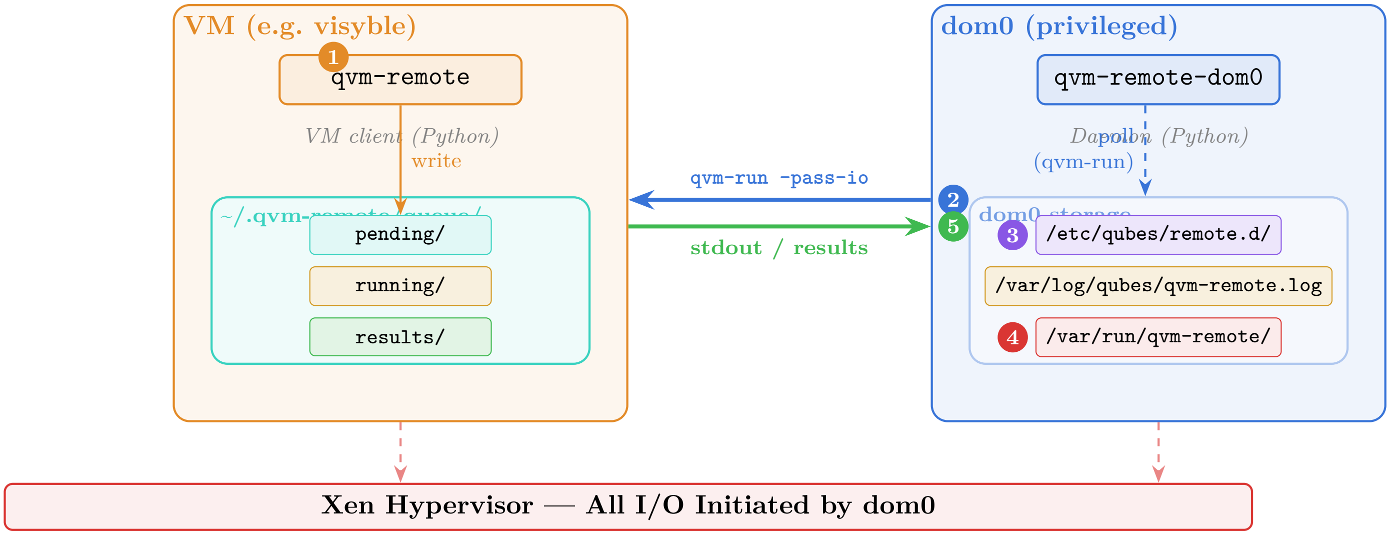

Figure 1: qvm-remote protocol architecture. **(1)** VM writes command + HMAC to `pending/`. **(2)** dom0 polls via `qvm-run -pass-io`. **(3)** Verifies HMAC against stored key. **(4)** Executes in sandboxed work directory. **(5)** Writes results back to VM. All I/O is initiated by dom0 (pull model).

3. **Authenticate** (dom0): For each command, reads `.auth`, recomputes HMAC, verifies with `hmac.compare_digest`. Rejects on mismatch.
4. **Execute** (dom0): Writes command to a 0700 work file, moves *cid* from `pending/` to `running/`, executes with `bash` under a timeout.
5. **Return** (dom0): Writes `.out`, `.err`, `.exit`, `.meta` to `results/`, removes from `running/`, appends to audit log.

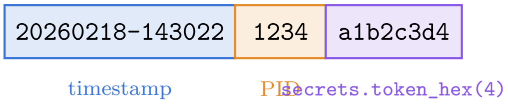

Figure 2: Command ID structure. Combines wall-clock time, process ID, and cryptographic randomness for uniqueness and non-replayability.

3.4 The "No Auto-Start" Property

A critical detail: the daemon uses `qvm-run -no-autostart`. If an authorized VM is powered off, it is silently skipped. The daemon never starts a VM as a side effect of polling—this prevents a compromised configuration file from causing VM launches.

4 Authentication Scheme

4.1 Key Management

Each VM-dom0 pair shares a 256-bit symmetric key k:

- **VM**: `~/.qvm-remote/auth.key` (mode 0600). Generated with `secrets.token_hex(32)`.
- **dom0**: `/etc/qubes/remote.d/`*`vm`*`.key` (mode 0600, directory 0700).

Key exchange is out-of-band: the user runs `qvm-remote key gen` in the VM, then copies the key to dom0 via `qvm-remote-dom0 authorize`. The key itself never traverses the queue protocol.

4.2 Per-Command Tokens

For each command with identifier *cid*, the VM computes:

$$\tau = \text{HMAC-SHA256}(k, \mathit{cid}) \tag{1}$$

and writes τ (as a hex string) to `pending/`*`cid`*`.auth`. dom0 recomputes τ' from the stored key and verifies:

$$\text{valid} \iff \texttt{hmac.compare\_digest}(\tau, \tau') \tag{2}$$

using Python's constant-time comparison to prevent timing side-channel attacks.

4.3 Security Properties

Table 1: Authentication properties and their guarantees.

Property	Guarantee
Forgery resistance	$\Pr[\text{forge}] \leq 2^{-256}$ per attempt. Key is never transmitted; only HMAC tokens appear in the queue.
Replay resistance	Each *cid* includes cryptographic randomness (`secrets.token_hex(4)`) and a timestamp. dom0 processes and deletes each *cid* exactly once.
Cross-VM isolation	Per-VM keys. Compromising VM A's key yields $\Pr[\text{forge}_B] = 2^{-256}$ for VM B.
Timing resistance	`hmac.compare_digest` runs in constant time regardless of input.

4.4 Brute-Force Analysis

The key space is 2^{256}. At 10^{12} HMAC computations per second (exceeding any current hardware), exhaustive search requires:

$$\frac{2^{256}}{10^{12} \times 365.25 \times 86400} \approx 3.7 \times 10^{57} \text{ years} \quad (3)$$

No retry limiting or lockout is needed—the key space makes brute force mathematically irrelevant.

5 Security Analysis

qvm-remote deliberately weakens the Qubes isolation model by granting VMs the ability to execute commands in dom0. This section analyzes the resulting attack surface and mitigations.

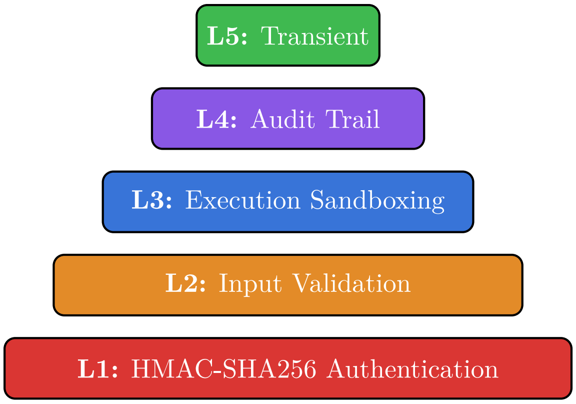
L5: Transient
L4: Audit Trail
L3: Execution Sandboxing
L2: Input Validation
L1: HMAC-SHA256 Authentication

Figure 3: Defense-in-depth: five security layers. Layer 5 (transient by default) means the service stops on reboot unless explicitly enabled.

5.1 Layer 1: HMAC-SHA256 Authentication

Every command must carry a valid HMAC token. Without the VM's 256-bit key, an attacker cannot forge a token (Section 4).

5.2 Layer 2: Input Validation

Before execution, dom0 validates:

- **Non-empty**: Empty commands are rejected.
- **Size limit**: Commands >1 MiB are rejected.
- **No binary**: Commands containing null bytes or excessive control characters are rejected (`has_binary_content()`).
- **Validation-before-write**: All checks occur before the command is written to dom0's filesystem.

5.3 Layer 3: Execution Sandboxing

Validated commands are written to `/var/run/qvm-remote/` (a `RuntimeDirectory` with mode 0700) as temporary scripts (mode 0700), executed via `subprocess.run(["bash", path])` with a 300-second timeout, and immediately deleted afterward.

5.4 Layer 4: Audit Trail

Every command is logged on both sides:

- **dom0**: `/var/log/qubes/qvm-remote.log` with categories (AUTH-OK, AUTH-FAIL, EXEC, DONE, TIMEOUT).
- **VM**: `~/.qvm-remote/audit.log` plus full output in `history/YYYY-MM-DD/`*`cid`*`/`.

The web UI (`qvm-remote-webui`) provides real-time log viewing with color-coded categories and regex filtering.

5.5 Layer 5: Transient by Default

The systemd service is installed but *not enabled*. On reboot, the daemon does not start unless the administrator explicitly runs `qvm-remote-dom0 enable` and types "yes" to an interactive risk warning. This prevents forgotten services from persisting.

5.6 Threat Model

Table 2: Threat model: attacks and mitigations.

Attack	Layer	Mitigation
Forge command	L1	HMAC-SHA256 (2^{256})
Replay command	L1	Unique *cid* + delete
Binary injection	L2	`has_binary_content()`
Command bomb	L2	1 MiB size limit
Fork bomb / hang	L3	300 s timeout
Privilege escalation	L3	Already root (known risk)
Undetected abuse	L4	Dual-sided audit + WebUI
Forgotten service	L5	Transient by default
Cross-VM attack	L1	Per-VM keys
Key theft	—	File permissions (0600)

6 Implementation

6.1 Design Constraints

qvm-remote targets two distinct environments:

- **dom0**: Fedora-based, minimal package set. No `pip`, limited repository. Python 3.8+ available.
- **VM**: Any Linux (Fedora, Debian, Arch). User-space installation via `make install-vm`.

This dictates the "stdlib only" constraint: no external Python packages. The entire implementation uses only `hashlib`, `hmac`, `secrets`, `subprocess`, `pathlib`, `os`, `sys`, `time`, and `signal`.

6.2 Component Sizes

Table 3: Implementation: single-file components.

Component	Location	Lines
Dom0 daemon	`dom0/qvm-remote-dom0`	685
VM client	`vm/qvm-remote`	423
Web UI	`dom0/qvm-remote-webui`	390
Test suite	`test/test_qvm_remote.py`	280+
Total		~1,800

6.3 Packaging and Distribution

- **Fedora RPM**: Separate `-dom0` and `-vm` packages. Built in a Fedora 41 Docker container (`make docker-rpm`).
- **Arch Linux**: `PKGBUILD` for the VM client.
- **Qubes Builder v2**: `.qubesbuilder` manifest for `qb -c qvm-remote package fetch prep build`.
- **Salt**: Formula in `salt/` for automated VM creation and daemon deployment.

6.4 CI Pipeline

GitHub Actions runs four parallel jobs:

1. Syntax check + unit tests (`make check && make test`)
2. Docker install test (Fedora 41 container)
3. Dom0 simulation E2E (mock `qvm-run`, `qvm-check`)
4. Arch Linux client test

7 Evaluation

7.1 Latency

We measured end-to-end command latency (VM client to result receipt) for commands of varying complexity.

Table 4: Command latency (median of 50 runs on Qubes 4.3).

Command	Latency	Overhead
`echo ok` (baseline)	48 ms	—
`hostname`	52 ms	+4 ms
`qvm-ls`	310 ms	+262 ms
`qvm-ls -format json`	380 ms	+332 ms

The qvm-remote overhead (polling + HMAC verification + file I/O) is ∼48 ms. The remainder is the command's own execution time, which is identical to running it directly in dom0.

7.2 Polling Efficiency

The daemon polls at a 1-second interval. Average discovery latency for a new command is 500 ms (uniform distribution over the interval). For interactive use, this is imperceptible. For batch automation, commands can be pre-queued and processed sequentially.

7.3 Comparison with Alternatives

Table 5: Comparison with alternative dom0 access methods.

Property	Manual	Qrexec	qvm-remote
Ad-hoc commands	✓	×	✓
Authentication	Physical	Policy	HMAC-256
Audit trail	×	Partial	✓
Multi-VM	✓	Per-policy	✓
Input validation	Human	×	✓
Timeout control	Manual	×	✓
Scriptable	×	✓	✓
Pull model	N/A	×	✓

8 Migration and Compatibility

qvm-remote 1.0 is a complete rewrite of `qubes-remote` v0.x (bash). Key changes:

- Language: bash → Python 3 (type hints, structured error handling).
- Authentication: none → HMAC-SHA256.
- Naming: `qubes-remote` → `qvm-remote` (follows official `qvm-*` conventions).
- Data directory: `~/.qubes-remote/` → `~/.qvm-remote/` (auto-migrated).
- Config variables: `QUBES_REMOTE_VMS` → `QVM_REMOTE_VMS` (backward compatible).

RPM packages use `Obsoletes:` directives for clean upgrades. The first run of either component auto-migrates the data directory.

9 Related Work

Qubes OS qrexec [1] provides inter-VM RPC via Xen shared memory. Unlike qvm-remote, each qrexec service requires a pre-installed handler and static policy, making it unsuitable for ad-hoc commands.

Qubes Split GPG [2] uses qrexec to isolate GPG keys. qvm-remote generalizes the "isolated execution" pattern to arbitrary commands.

qubes-tunnel [3] provides VPN tunneling through Qubes VMs. Like qvm-remote, it bridges the dom0-VM boundary but focuses on network rather than command execution.

Ansible over Qubes [4] uses custom connection plugins to manage Qubes VMs. qvm-remote provides the *reverse* direction (VM → dom0) that Ansible cannot address without dom0 agent installation.

qubes-claw [5] builds on qvm-remote to provide airgapped AI agent administration, demonstrating that the queue protocol can bootstrap more complex infrastructure.

10 Conclusion

qvm-remote demonstrates that the tension between Qubes OS's dom0 isolation and practical administration can be resolved with a carefully designed pull-model protocol. By keeping all I/O initiation in dom0, using HMAC-SHA256 for per-command authentication, and defaulting to transient operation, the system provides the convenience of SSH-like access while maintaining five independent security layers.

The implementation is minimal (∼1,800 lines of stdlib-only Python), packaged for three distribution channels, and tested through four CI pipelines. As a building block, it has already enabled the qubes-claw airgapped AI infrastructure, demonstrating its utility beyond simple administration.

Availability: https://github.com/GabrieleRisso/qvm-remote

References

[1] J. Rutkowska and R. Wojtczuk. Qubes OS architecture. *Invisible Things Lab*, 2010. https://www.qubes-os.org/doc/architecture/

[2] Qubes OS Project. Split GPG. *Qubes OS Documentation*, 2023. https://www.qubes-os.org/doc/split-gpg/

[3] tasket. qubes-tunnel: VPN setup for Qubes OS. GitHub, 2021. https://github.com/tasket/Qubes-vpn-support

[4] Ansible Community. Qubes OS connection plugin. *Ansible Galaxy*, 2024.

[5] G. Risso. qubes-claw: Secure, isolated AI agent infrastructure on Qubes OS. GitHub, 2026. https://github.com/GabrieleRisso/qubes-claw